Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)
Stride-1 reference patterns are good (spatial locality)

Examples:
- cold cache, 4-byte words, 4-word cache blocks
- int sumarrayrows(int a[M][N])
  int i, j, sum = 0;
  for (i = 0; i < M; ++i)
    for (j = 0; j < N; ++j)
      sum += a[i][j];
  return sum;

- int sumarraycols(int a[M][N])
  int i, j, sum = 0;
  for (j = 0; j < N; ++j)
    for (i = 0; i < M; ++i)
      sum += a[i][j];
  return sum;

The Memory Mountain

Read throughput (read bandwidth)
- Number of bytes read from memory per second (MB/s)

Memory mountain
- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.

Memory Mountain Test Function

/* The test function */
void test(int elems, int stride) {
  volatile int sink;
  int i, result = 0;
  for (i = 0; i < elems; i += stride)
    result += data[i];
  sink = result; /* So compiler doesn’t optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
  double cycles;
  int elems = size / sizeof(int);
  test(elems, stride); /* warm up the cache */
  cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
  return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}

Memory Mountain Main Routine

/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)
int data[MAXELEMS]; /* The array we’ll be traversing */

int main() {
  int size; /* Working set size (in bytes) */
  int stride; /* Stride (in array elements) */
  double Mhz; /* Clock frequency */
  int main() {
    /* Initialize each element in data to 1 */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
      for (stride = 1; stride <= MAXSTRIDE; stride++) {
        printf("%ld", run(size, stride, Mhz));
        printf(">%d", run(size, stride, Mhz));
        exit(0);
    }
  }
}
Pentium 4 – 2.4 GHz

Memory performance (stride = 6)

Ridges of Temporal Locality
Slice through the memory mountain with stride=1
- illuminates read throughputs of different caches and memory

A Slope of Spatial Locality
Slice through memory mountain with size=256KB
- shows cache block size.

Matrix Multiplication Example

Description:
- Multiply N x N matrices
- O(N^3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

Major Cache Effects to Consider
- Total cache size
- Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
- Exploit spatial locality

/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
Miss Rate Analysis for Matrix Multiply

Assume:
- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:
- Look at access pattern of inner loop

![Diagram of matrix access pattern]

Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:
- A: 0.25
- B: 1.0
- C: 0.0

Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per Inner Loop Iteration:
- A: 0.25
- B: 1.0
- C: 0.0

Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per Inner Loop Iteration:
- A: 0.0
- B: 0.25
- C: 0.25

Matrix Multiplication (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per Inner Loop Iteration:
- A: 0.0
- B: 0.25
- C: 0.25
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

- **ijk** (& jik): 2 loads, 0 stores
  - misses/iter = 1.25
  - kij (@ ikj):
    - 2 loads, 1 store
    - misses/iter = 0.5
  - kji (@ jki):
    - 2 loads, 1 store
    - misses/iter = 2.0

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.
- Code scheduling matters, too.

Improving Temporal Locality by Blocking

**Example:** Blocked matrix multiplication
- "block" (in this context) does not mean "cache block".
- Instead, it means a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

```
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++)
                sum = 0.0;
                for (k=kk; k < min(kk+bsize,n); k++)
                    sum += a[i][k] * b[k][j];
            c[i][j] += sum;
    }
}
```

Key idea: Sub-blocks (& A) can be treated just like scalars.

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix} =
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Blocked Matrix Multiply (bijk)

```
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++)
                sum = 0.0;
                for (k=kk; k < min(kk+bsize,n); k++)
                    sum += a[i][k] * b[k][j];
            c[i][j] += sum;
    }
}
```
**Blocked Matrix Multiply Analysis**

- Innermost loop pair multiplies a $1 \times \text{bsize}$ sliver of $A$ by a $\text{bsize} \times \text{bsize}$ block of $B$ and accumulates into a $1 \times \text{bsize}$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

**Pentium Blocked Matrix Multiply Performance**

Blocking ($bijk$ and $bikj$) improves performance by a factor of two over unblocked versions ($ijk$ and $jik$)

- relatively insensitive to array size.

![Graph showing Pentium Blocked Matrix Multiply Performance](image)

**Concluding Observations**

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)